

Если вы видите что-то необычное, просто сообщите мне.

Type Families in Haskell

Welcome to the conclusion of our series on Haskell data types! We've gone over a lot of things in this series that demonstrated Haskell's simplicity. We compared Haskell against other languages where we saw more cumbersome syntax. In this final part, we'll see something a bit more complicated though. We'll do a quick exploration of the idea of type families. We'll start by tracing the evolution of some related type ideas, and then look at a quick example.

This is a beginner series, but the material in this final part will be a bit more complicated. If the code examples are confusing, it'll help to read our Monads Series first! But if you're just starting out, we've got plenty of other resources to help you out! Take a look at our Getting Started Checklist or our Liftoff Series!

You can follow along with these code examples in our Github Repository! Just take a look at the Type Families module!

DIFFERENT KINDS OF TYPE HOLES

In this series so far, we've seen a couple different ways to "plug in a hole", as far as a type or class definition goes. In the third part of this series we explored parametric types. These have type variables as part of their definition. We can view each type variable as a hole we need to fill in with another type.

Then in the fourth part, we explored the concept of typeclasses. For any instance of a typeclass, we're plugging in the holes of the function definitions of that class. We fill in each hole with an implementation of the function for that particular type.

In this last part, we're going to combine these ideas to get type families! A type family is an enhanced class where one or more of the "holes" we fill in is actually a type! This allows us to associate different types with each other. The result is that we can write special kinds of polymorphic functions.

A BASIC LOGGER

First, here's a contrived example to use through this article. We want to have a logging typeclass. We'll call it `MyLogger`. We'll have two main functions in this class. We should be able to get all the messages in the log in chronological order. Then we should be able to log a new message, which will naturally affect the logger type. A first pass at this class might look like this:

```
class MyLogger logger where
  prevMessages :: logger -> [String]
  logString :: String -> logger -> logger
```

We can make a slight change that would use the `State` monad instead of passing the logger as an argument:

```
class MyLogger logger where
  prevMessages :: logger -> [String]
  logString :: String -> State logger ()
```

But this class is deficient in an important way. We won't be able to have any effects associated with our logging. What if we want to save the log message in a database, send it over network connection, or log it to the console? We could allow this, while still keeping `prevMessages` pure like so:

```
class MyLogger logger where
  prevMessages :: logger -> [String]
  logString :: String -> StateT logger IO ()
```

Now our `logString` function can use arbitrary effects. But this has the obvious downside that it forces us to introduce the `IO` monad places where we don't need it. If our logger doesn't need `IO`, we don't want it. So what do we do?

USING A MONAD

One place we can start is to make the logger itself the monad! Then getting the previous messages will be a simple matter of turning that function into an effect. And then we won't necessarily be bound to the State monad:

```
class (Monad m) => MyLoggerMonad m where
  prevMessages :: m [String]
  logString :: String -> m ()
```

But now suppose we want to give our user the flexibility to use something besides a list of strings as the "state" of the message system. Maybe they also want timestamps, or log file information. We want to tie this type to the monad itself, so we can use it in different function signatures. That is, we want to fill in a "hole" in our class instance with a particular type. How do we do this?

TYPE FAMILY BASICS

One answer is to make our class a type family. We do this with the type keyword in the class definition. First, we need a few language pragmas to allow this:

```
{-# LANGUAGE AllowAmbiguousTypes #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
{-# LANGUAGE TypeFamilies #-}
```

Now we'll make a type within our class that refers to the state we'll use. We have to describe the "kind" of the type with the definition. Since our state is an ordinary type that doesn't take a parameter, its kind is *. Here's what our definition looks like:

```
class (Monad m) => MyLoggerMonad m where
  type LogState m :: *
  retrieveState :: m (LogState m)
  logString :: String -> m ()
```

Instead of returning a list of strings all the time, `retrieveState` will produce whatever type we assign as the `LogState`. Since our state is more general now, we'll call the function `retrieveState` instead of `prevMessages`.

A SIMPLE INSTANCE

Now that we have our class, let's make a monad that implements it! Our first example will be simple, wrapping a list of strings with `State`, without using IO:

```
newtype ListWrapper a = ListWrapper (State [String] a)
deriving (Functor, Applicative, Monad)
```

We'll assign `[String]` to be the stateful type. Then retrieving that is as simple as using `get`, and adding a message will push it at the head of the list.

```
instance MyLoggerMonad ListWrapper where
  type LogState ListWrapper = [String]
  retrieveState = ListWrapper get
  logString msg = ListWrapper $ do
    prev <- get
    put (msg : prev)
```

A function using this monad could call the `logString` function, and retrieve its state:

```
produceStringsList :: ListWrapper [String]
produceStringsList = do
  logString "Hello"
  logString "World"
  retrieveState
```

To run this monadic action, we'd have to get back to the basics of using the `State` monad. (Again, our [Monads series](#) explains those details in more depth). But at the end of the day we can produce a pure list of strings.

```
listWrapper :: [String]
listWrapper = runListWrapper produceStringsList
```

```
runListWrapper :: ListWrapper a -> a
runListWrapper (ListWrapper action) = evalState action []
```

USING IO IN OUR INSTANCES

Now we can make a couple different versions of this logger that actually use IO. In our first example, we'll use a map instead of a list as our "state". Each new message will have a timestamp associated with it, and this will require IO. When we log a string, we'll get the current time and store the string in the map with that time.

```
type TimeMsgMap = M.Map UTCTime String
newtype StampedMessages a = StampedMessages (StateT TimeMsgMap IO a)
  deriving (Functor, Applicative, Monad)

instance MyLoggerMonad StampedMessages where
  type LogState StampedMessages = TimeMsgMap
  retrieveState = StampedMessages get
  logString msg = StampedMessages $ do
    ts <- lift getCurrentTime
    lift (print ts)
    prev <- get
    put (M.insert ts msg prev)
```

And then we can make another version of this that logs the messages in a file. The monad will use ReaderT to track the name of the file, and it will open it whenever it needs to log a message or produce more output:

```
newtype FileLogger a = FileLogger (ReaderT FilePath IO a)
  deriving (Functor, Applicative, Monad)

instance MyLoggerMonad FileLogger where
  type LogState FileLogger = [String]
  retrieveState = FileLogger $ do
    fp <- ask
    (reverse . lines) <$> lift (readFile fp)
  logString msg = FileLogger $ do
```

```
lift (putStrLn msg)          -- Print message
fp <- ask                    -- Retrieve log file
lift (appendFile fp (msg ++ "\n")) -- Add new message
```

We can also use the IO to print our message to the console while we're at it.

#USING OUR LOGGER By defining our class like this, we can now write a polymorphic function that will work with any of our loggers! Once we apply the constraint in our signature, we can use the LogState as another type in our signature!

```
useAnyLogger :: (MyLoggerMonad m) => m (LogState m)
useAnyLogger = do
  logString "Hello"
  logString "World"
  logString "!"
  retrieveState

runListGeneric :: [String]
runListGeneric = runListWrapper useAnyLogger

runStampGeneric :: IO TimeMsgMap
runStampGeneric = runStampWrapper useAnyLogger
```

This is awesome because our code is now abstracted away from the needed effects. We could call this with or without the IO monad.

COMPARING TO OTHER LANGUAGES

When it comes to effects, Haskell's type system often makes it more difficult to use than other languages. Arbitrary effects can happen anywhere in Java or Python. Because of this, we don't have to worry about matching up effects with types.

But let's not forget about the benefits of Haskell's effect system! For all parts of our code, we know what effects we can use. This lets us determine at compile time where certain problems can arise.

Type families give us the best of both worlds! They allow us to write polymorphic code that can work either with or without IO effects. This is really cool, especially whenever you want to have different setups for testing and development.

Haskell is a clear winner when it comes to associating types with one another and applying compile-time constraints on these relationships. In C++ it is possible to get this functionality, but the syntax is very painful and out of the ordinary. In Haskell, type families are a complex topic to understand. But once you've wrapped your head around the concept, the syntax is actually fairly intuitive. It springs naturally from the existing mechanisms for typeclasses, and this is a big plus.

CONCLUSION

That's all for our series on Haskell's data system! We've now seen a wide range of elements, from the simple to the complex. We compared Haskell against other languages. Again, the simplicity with which one can declare data in Haskell and use it polymorphically was a key selling point for me!

Hopefully this series has inspired you to get started with Haskell if you haven't already! Download our Getting Started Checklist or read our Liftoff Series to get going!

And don't forget to try this code out for yourself on Github! Take a look at the Type Families module for the code from this part!

Revision #1

Created 11 March 2022 06:09:22 by gasick

Updated 11 March 2022 17:11:17 by gasick